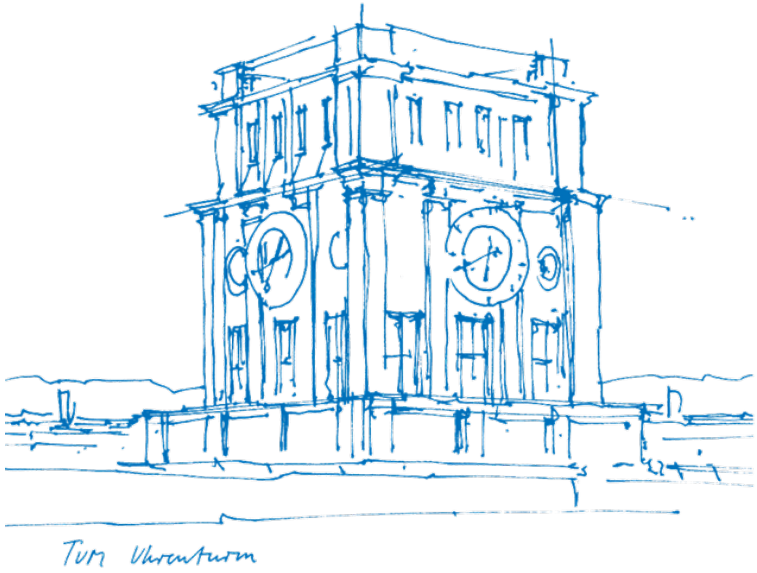
# Sparse Linear Algebra in CUDA

HPC - Algorithms and Applications

Alexander Pöppl
Technical University of Munich
Chair of Scientific Computing

November 28th 2018



# Table of Contents


**Homework - Worksheet 2**

**Recap on Coalesced Memory Access**

**PageRank algorithm**

**Compressed Sparse Row (CSR) Kernels**

- Scalar kernel
- Vectorized kernel

# Last Tutorial

## Matrix Multiplication: Optimizations

- coalesced memory access
- overlapped memory access and computation

## Roofline Model

- Compares hardware and kernel performance limits
- Hints for optimizations through ceilings

# Assignment H3.2a - Vectorized CSR kernel

## CMake

- Experimental CMake support for this exercise
- CMake can create Projects for different platforms
  - Makefiles
  - Visual studio projects
- To use it:
  - Windows: Use cmake GUI
  - Linux/OSX: in Project folder:

    ```
    mkdir build; cd build; cmake ..; make
    ```

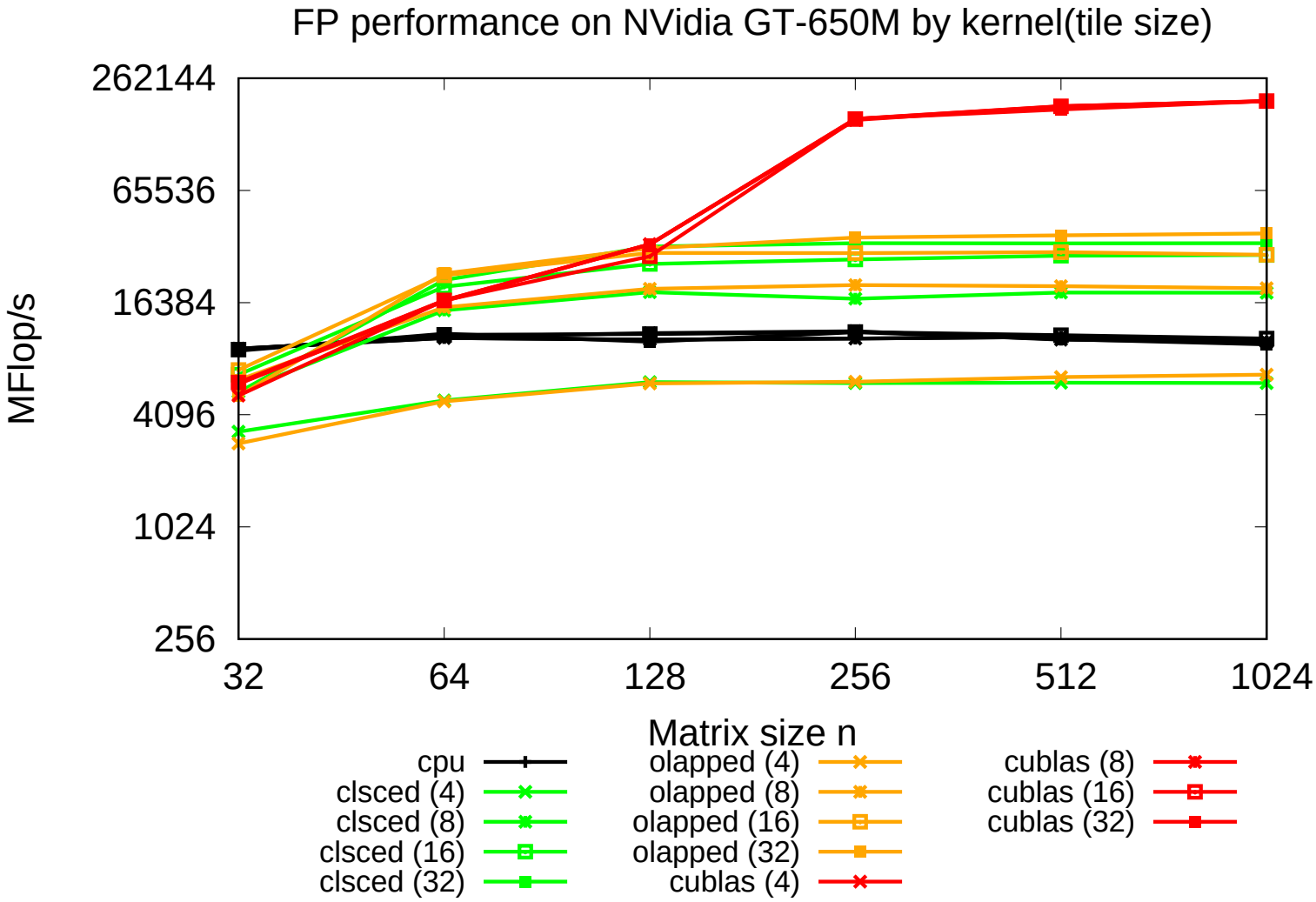
# Assignment H2.1a - Prefetching

```
__global__ void mm_o(float* Ad, float* Bd,
                     float* Cd, int n) {
  /** snip **/
  float Celem = 0;
  float Areg = Ad[ i*n + tx];
  float Breg = Bd[ ty*n + k];

  for(int m=1; m < n/TILE_SIZE; m++) {
    Ads[ty][tx] = Areg;
    Bds[ty][tx] = Breg;
    syncthreads();

    Areg = Ad[ i*n + m*TILE_SIZE+tx];
    Breg = Bd[ (m*TILE_SIZE+ty)*n + k];
    /** cont. **/
```

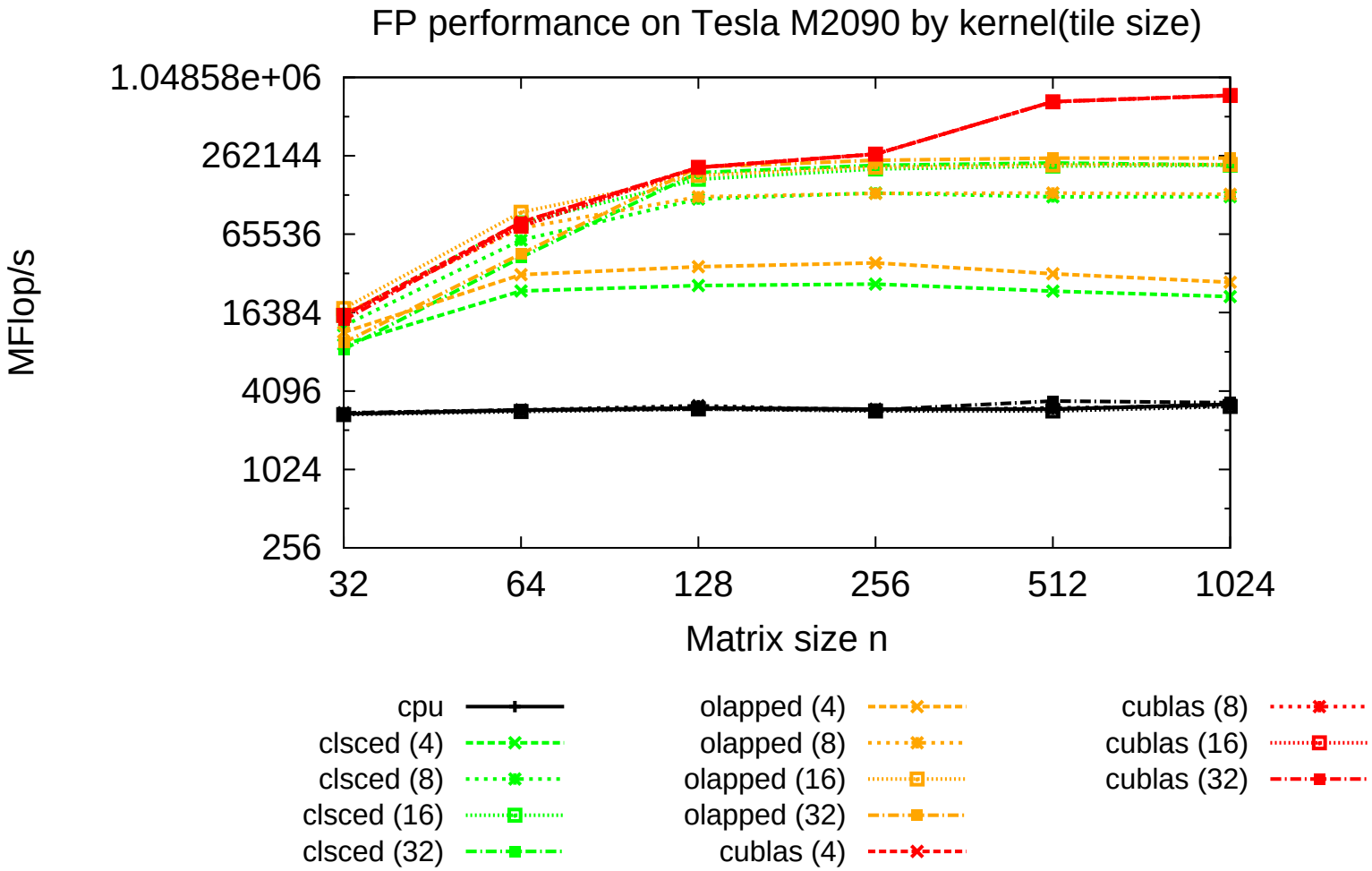
# Assignment H2.1a - Prefetching

```
    /** cont. **/
    for(int j=0; j<TILE_SIZE; j++) {
      Celem += Ads[ty][j]*Bds[j][tx];
    }
    syncthreads();
  };

  Ads[ty][tx] = Areg;
  Bds[ty][tx] = Breg;
  syncthreads();
  for(int j=0; j<TILE_SIZE; j++) {
    Celem += Ads[ty][j]*Bds[j][tx];
  }
  Cd[i*n+k] += Celem;
}
```

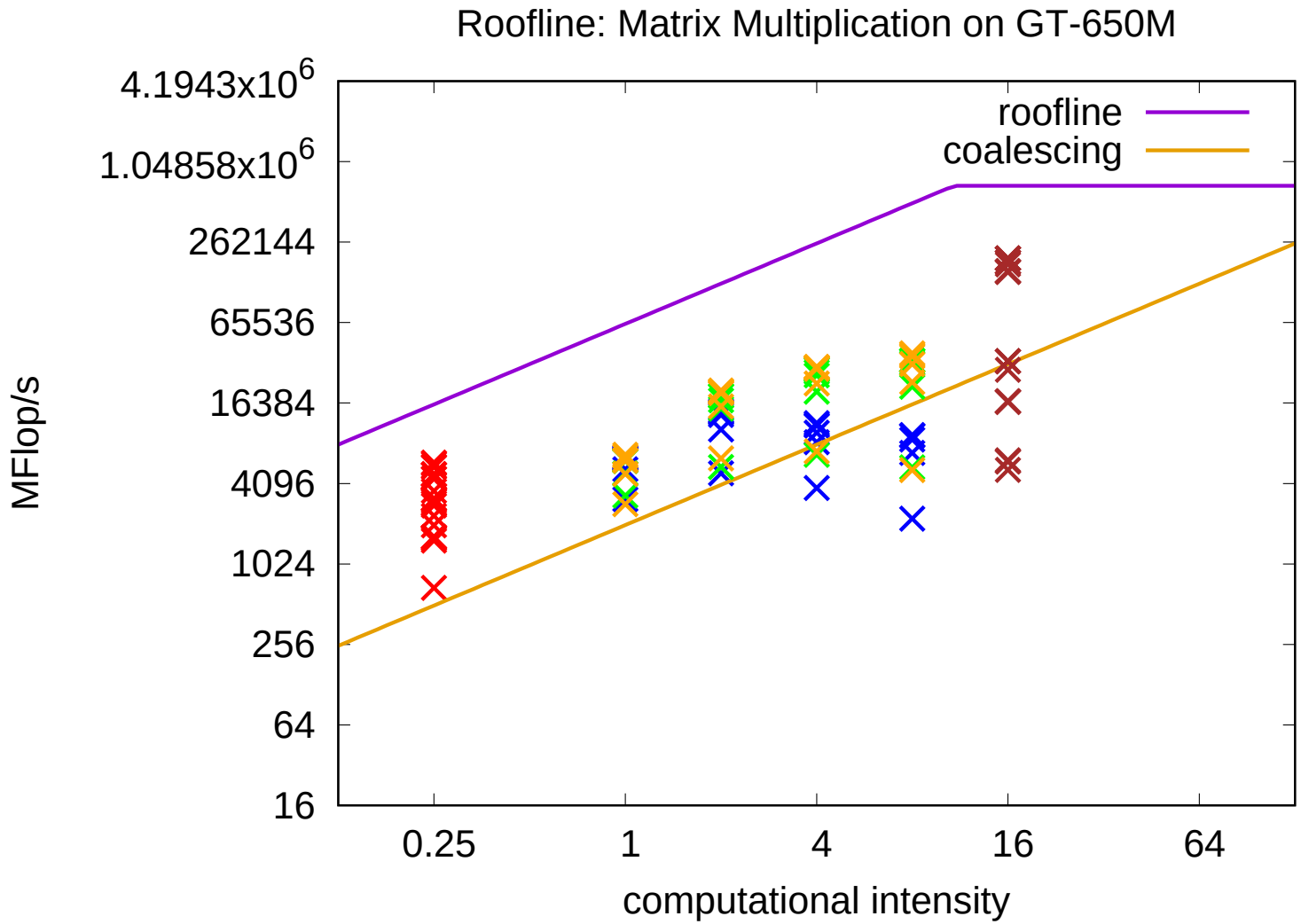
# Assignment H2.1b - Performance measurements



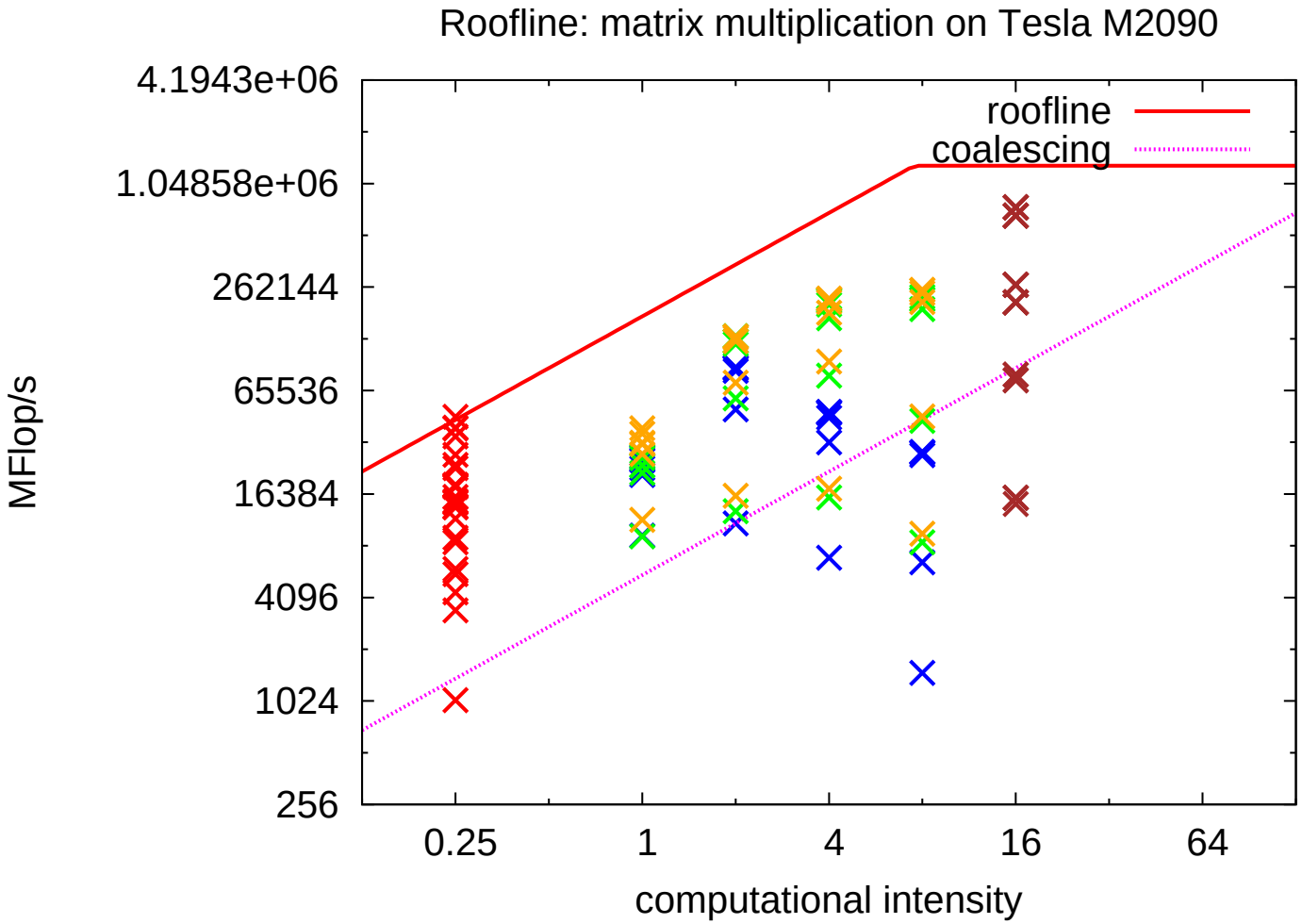
# Assignment H2.1b - Performance measurements



# Assignment H2.1b - Roofline Model



# Assignment H2.1b - Roofline Model



# Prefetching - a resource trade-off

Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290 (8192 Registers per SM)

- before prefetching 10 Registers per thread $\rightarrow 16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks

# Prefetching - a resource trade-off

Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290 (8192 Registers per SM)

- before prefetching 10 Registers per thread $\rightarrow 16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
- now 16 registers $\rightarrow 16 \times 16 \times 16 = 4096$ registers per block
- only 8192 registers per SM $\rightarrow$ only 2 active blocks per SM!

# Prefetching - a resource trade-off

Prefetching: no massive performance gain on all systems!
Example: NVidia Quadro NVS 290 (8192 Registers per SM)

- before prefetching 10 Registers per thread $\rightarrow 16 \times 16 \times 10 = 2560$ registers per block
- 8192 registers per SM $\rightarrow$ 3 active blocks
- now 16 registers $\rightarrow 16 \times 16 \times 16 = 4096$ registers per block
- only 8192 registers per SM $\rightarrow$ only 2 active blocks per SM!

Number of active threads

- before: 3 active blocks $\rightarrow$ 24 active warps $\rightarrow$ 768 active threads $\rightarrow$ optimal occupancy
- after: 2 active blocks $\rightarrow$ 16 active warps $\rightarrow$ 512 active threads $\rightarrow$ less parallelism, less latency hiding

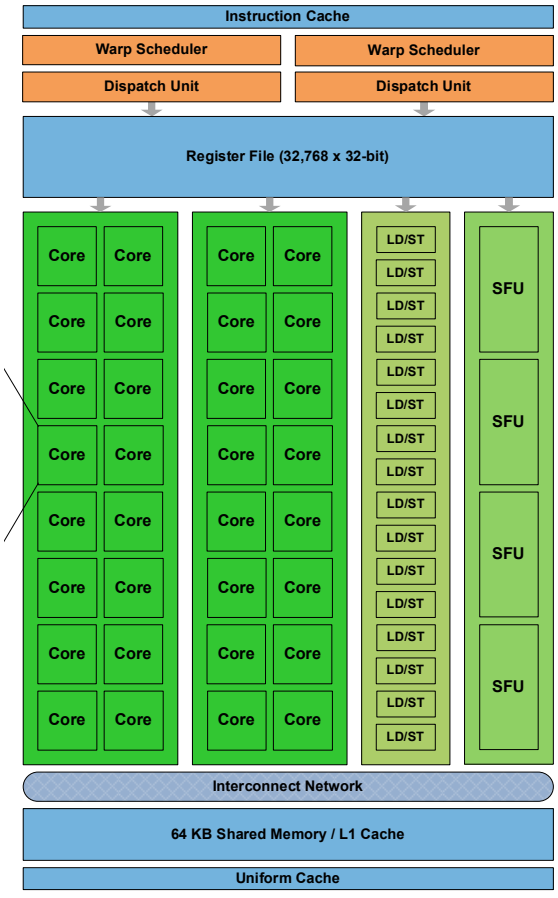
This may be different for your GPU!

# Towards High-Performance Matrix Multiplication

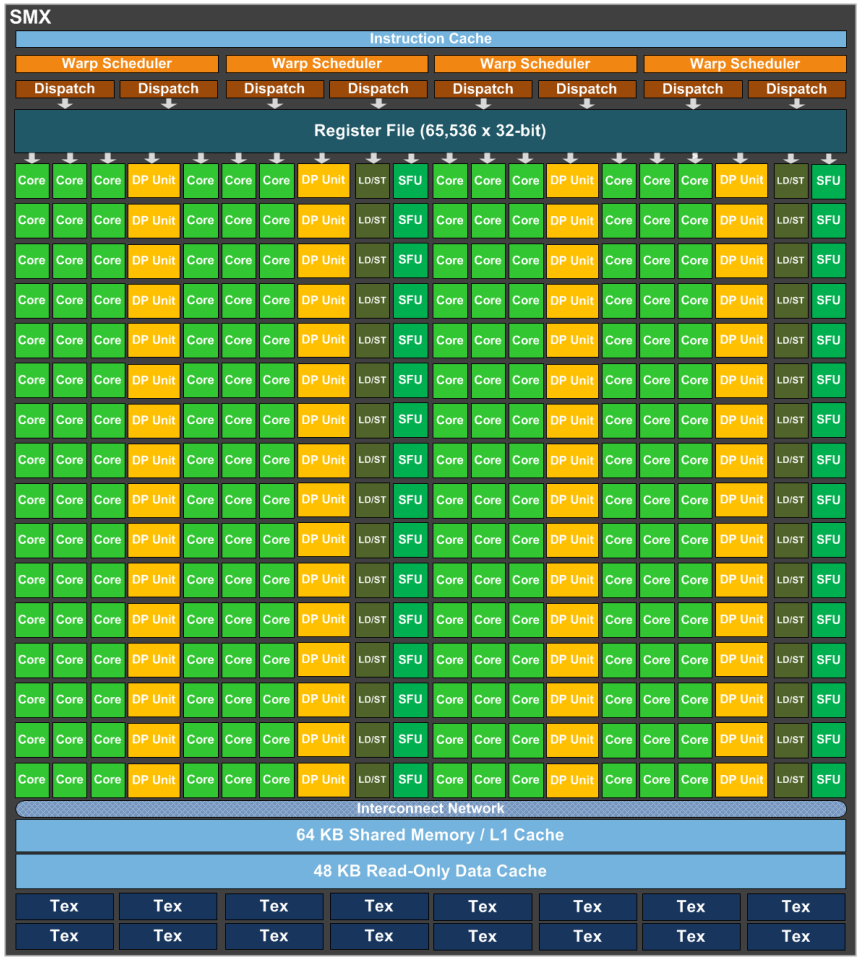
More options for optimization:

- loop unrolling (save loop instructions and address arithmetics)
- thread granularity: compute $1 \times 2$ or $1 \times 4$ blocks per thread (requires to load `Ads` or `Bds` only once)
- how do different optimizations interact with resource limitations (available registers, etc.)

# Recap: Coalescing




Fermi Streaming Multiprocessor (SM)




(source: NVIDIA – Fermi/Kepler Whitepapers)

# Recap: Coalescing

Hardware limitations:

- Fermi: 1 dispatcher per warp $\Rightarrow$ 1 concurrent instruction
- Kepler: 2 dispatchers per warp $\Rightarrow$ 2 concurrent instructions

Bottleneck:

- biggest LD/ST instruction can transfer 128 B chunk
- if threads in a warp access multiple 128 B chunks in global memory, multiple LD/ST instructions have to be dispatched for the warp.
- if all threads access the same 128 B chunk, a single LD/ST of size 128 B is enough for the warp.

# Recap: Coalescing

On this slide:

```
tx = threadIdx.x; ty = threadIdx.y; tz = threadIdx.z;
```

Coalescing:

- Ideally: each thread in a warp accesses the same 128 B chunk
- In order to achieve that, we must know which threads of a block are in a warp.
- CUDA uses row-major order for warp indices, so:

  ```
  warpID = ((tz * blockDim.y + ty) * blockDim.x + tx) / 32;
  ```

- Rule of thumb: Memory access `A[f(tx, ty, tz)]` to an array `float* A;` is coalesced, if little change to `tx` causes little change to `f(tx, ty, tz)`.

**Task:**

- Solve exercise T3.1 on your worksheet.
- Besides coalescing, no further hardware optimizations are assumed to happen

# Sparse Linear Algebra

Nathan Bell and Michael Garland
Efficient Sparse Matrix-Vector Multiplication on CUDA.
2008.

Goals:

- Large matrices in big applications always sparse
- Efficient treatment of sparsity
- Towards higher numbers of unknowns

# PageRank algorithm

**Input:** $\mathbf{B} \in \mathbb{R}^{n \times n}$ left-stochastic (non-negative and all column sums are 1), $\alpha \in (0, 1)$, $\epsilon > 0$

**Output:** $\mathbf{x} \in \mathbb{R}^n$ stochastic (non-negative and sum is 1) with $\mathbf{x} \approx \mathbf{Bx}$

$\mathbf{x}^{(0)} \leftarrow \frac{1}{n}\mathbf{e}$, where $\mathbf{e} = (1, 1, 1, ...)^T$;

$i \leftarrow 0$;

**repeat**

  $\mathbf{y}^{(i)} \leftarrow \mathbf{Bx}^{(i)}$;

  $\mathbf{x}^{(i+1)} \leftarrow \alpha\mathbf{y}^{(i)} + (1 - \alpha)\frac{1}{n}\mathbf{e}$;

  $i \leftarrow i + 1$;

**until** $||\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}|| < \epsilon$;

$\mathbf{x} \leftarrow \mathbf{x}^{(i)}$;

# Compressed Sparse Row (CSR)

CSR matrix-vector multiplication:

```
const int N;        // number of matrix rows
const int K;        // number of nonzero matrix entries
float a[K];         // array of nonzero matrix entries
float j[K];         // array of column indices
float start[N+1]; // array of row start indices
float x[N];         // input vector x
float y[N];         // result vector y

for(int i = 0; i < N; i++) {
  y[i] = 0;
  for(k = start[i]; k < start[i + 1]; k++) {
    y[i] += a[k] * x[j[k]];
  }
}
```

# Compressed Sparse Row (CSR) Kernel 1

```
__global__ void csr_matvec_s(start, j, a, x, y) {

  /** TODO **/

}
```

Task: Implement a CSR matrix-vector multiplication $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{Bx}$:

- Assign one thread to a matrix row
- Compute a row $\times$ vector product for each thread.
- Add the result to the output vector.

# Compressed Sparse Row (CSR) Kernel 1

Solution:

```
__global__ void csr_matvec_s(start, j, a, x, y) {
  int i = blockDim.x * blockIdx.x + threadIdx.x ;
  if (i < num_rows) {
    float dot = 0;
    int row_start = start[i];
    int row_end = start[i + 1];
    for (int k = row_start; k < row_end; k++) {
      dot += a[k] * x[j[k]];
    }

    y[i] += dot;
  }
}
```

# Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of `j` and `a`

**Example data:**

```
start                   [0 2 4 7 9]
```

**Access pattern to `j` and `a` by row / thread ID (0-3):**

```
k = row_start           [0   1   2     3  ]
k = row_start + 1       [  0   1   2     3]
k = row_start + 2       [            2    ]
```

# Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of `j` and `a`
- `x` is accessed randomly $\rightarrow$ **uncoalesced** access

**Example data:**

```
start                   [0 2 4 7 9]
```

**Access pattern to `j` and `a` by row / thread ID (0-3):**

```
k = row_start           [0   1   2     3  ]
k = row_start + 1       [  0   1   2     3]
k = row_start + 2       [            2    ]
```

# Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of `j` and `a`
- `x` is accessed randomly $\rightarrow$ **uncoalesced** access
- **uncoalesced** memory access to `j` and `a`,
  **coalesced** access to `start` and `y`

**Example data:**

```
start                    [0 2 4 7 9]
```

**Access pattern to `j` and `a` by row / thread ID (0-3):**

```
k = row_start            [0   1   2     3  ]
k = row_start + 1        [  0   1   2     3]
k = row_start + 2        [            2    ]
```

# Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of `j` and `a`
- `x` is accessed randomly $\rightarrow$ **uncoalesced** access
- **uncoalesced** memory access to `j` and `a`,
  **coalesced** access to `start` and `y`
- non-uniform distribution of nonzeros may lead to serialization

**Example data:**

```
start                    [0 2 4 7 9]
```

**Access pattern to `j` and `a` by row / thread ID (0-3):**

```
k = row_start            [0   1   2     3  ]
k = row_start + 1        [  0   1   2     3]
k = row_start + 2        [            2    ]
```

# Compressed Sparse Row (CSR) Kernel 2

Idea: each **warp** does a row $\times$ vector multiplication.

Requires the following steps:

- Assign one warp to a matrix row
- Allocate a shared array `vals[]` for the partial results of a block
- Compute one row $\times$ vector product in a loop. This time, parallelize the loop over all 32 threads in the warp. Take care that access to the arrays `j` and `a` is coalesced.
- Use a reduction of some kind (ideally: binary fan-in) to add up the partial sums in `vals[]` and add the output to the result vector.

$\rightarrow$ Homework!